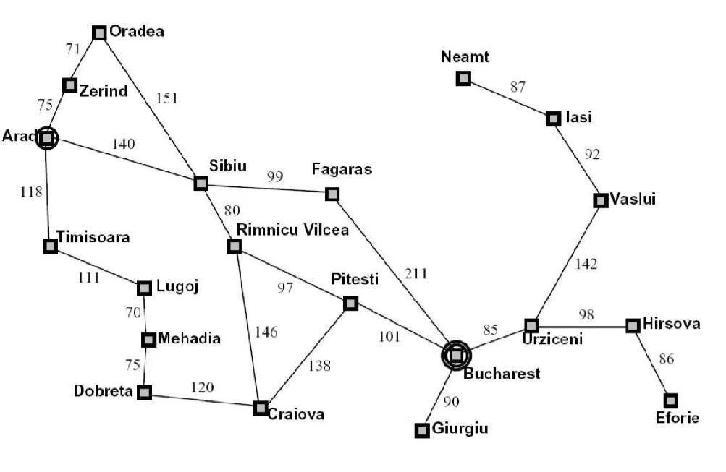
**DFS、BFS、UCS、A\*、GS贪心这几个算法我都将采用Romania这个例子来展开进行：**

**具体问题描述：**

****

1. **DFS**
2. **问题描述及DFS介绍：**

DFS：深度优先探索，“一条道走到黑”

用DFS实现Romania从Arad到Bucharest的遍历，将经过的城市（node）标红。

1. **设计思路（伪代码）：**

FUNCTION DFS(graph, start, end, visited, path)：

# 输入：图graph, 起始节点start, 目标节点end, 已访问集合visited, 当前路径path

# 输出：从start到end的路径，若无解返回空列表

visited.add(start) # 标记当前节点为已访问

path.append(start) # 将当前节点加入路径

IF start == end THEN # 终止条件：到达目标节点

RETURN path # 返回当前路径

FOR EACH neighbor IN graph[start] DO

IF neighbor NOT IN visited THEN

result ← DFS(graph, neighbor, end, visited, path) # 递归搜索

IF result ≠ [] THEN

RETURN result # 找到有效路径则立即返回

path.remove(start) # 回溯：移除当前节点

RETURN [] # 返回空列表表示无解

END FUNCTION

# 主程序

PROCEDURE MAIN()：

# 定义图结构（邻接表）

graph ← {

'Arad': ['Zerind', 'Timisoara', 'Sibiu'],

'Zerind': ['Oradea', 'Arad'],

# ... 其他节点定义同原数据

}

start ← 'Arad' # 设置起点

end ← 'Bucharest' # 设置终点

visited ← EMPTY\_SET # 初始化已访问集合

path ← EMPTY\_LIST # 初始化路径

final\_path ← DFS(graph, start, end, visited, path) # 调用DFS

IF final\_path ≠ [] THEN

PRINT "找到路径：", final\_path

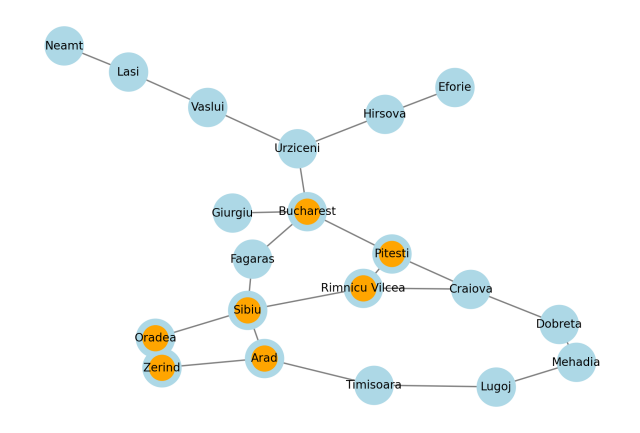
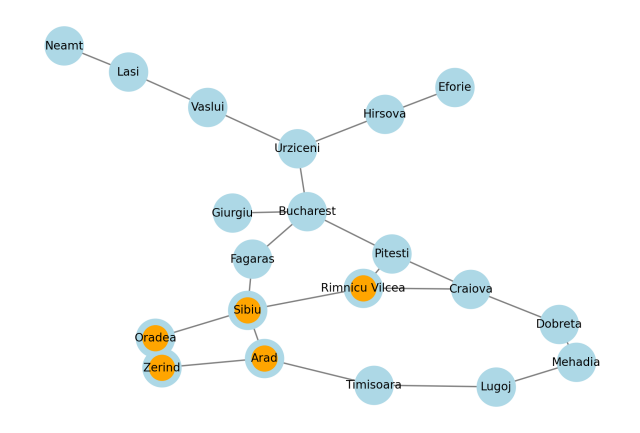
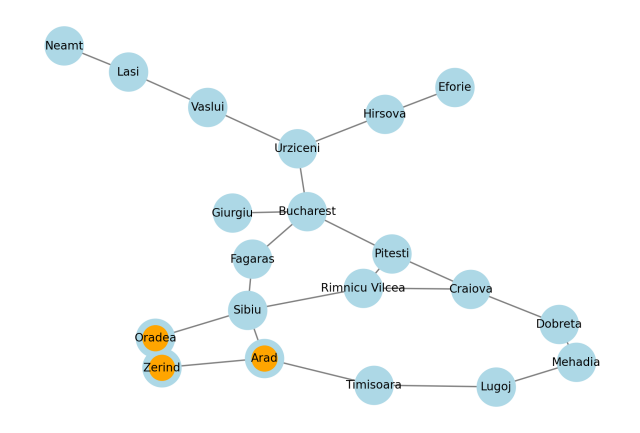
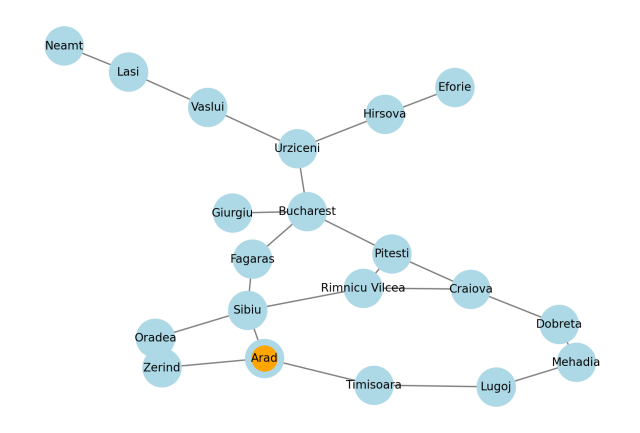
ELSE

PRINT "无可行路径"

END PROCEDURE

1. **实验结果：（部分中间图片省略，下同）**

可视化截图如下：



1. **BFS（广度优先探索）**
2. **问题描述以及BFS理解：**

BFS：广度优先探索，逐层扩展搜索，先访问所有相邻节点，再访问下一层节点。每次都在扩展自己的边缘，“广撒网”；BFS适用于无权图中最短路径的查找。

用BFS实现Romania从Arad到Bucharest的路径搜索，能找到最短路径。

1. **伪代码：**

FUNCTION BFS\_PATH(graph, start, end):

# 初始化

queue = EMPTY\_QUEUE() # 创建空队列用于BFS

visited = EMPTY\_SET() # 记录已访问节点

parent = EMPTY\_DICTIONARY() # 记录节点的前驱（用于回溯路径）

queue.ENQUEUE(start) # 将起点加入队列

visited.ADD(start) # 标记起点为已访问

parent[start] = NULL # 起点的前驱设为空

# 开始BFS遍历

WHILE queue IS NOT EMPTY:

current = queue.DEQUEUE() # 取出队列头部节点

# 检查是否到达终点

IF current == end:

path = EMPTY\_LIST()

node = end

# 回溯构建路径

WHILE node IS NOT NULL:

path.APPEND(node)

node = parent[node]

RETURN REVERSE(path) # 反转路径得到从起点到终点的顺序

# 遍历邻居节点

FOR neighbor IN graph[current]:

IF neighbor NOT IN visited:

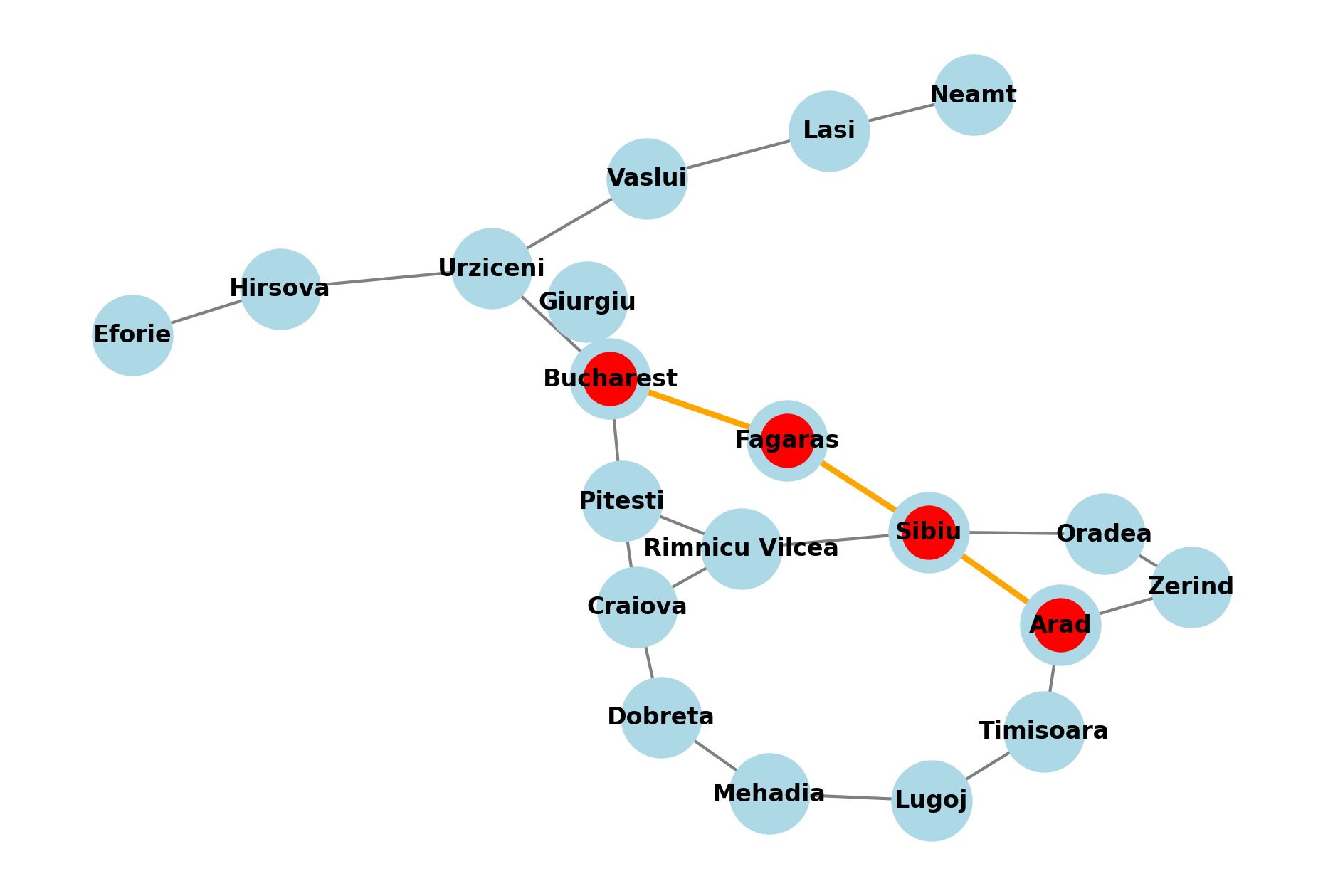
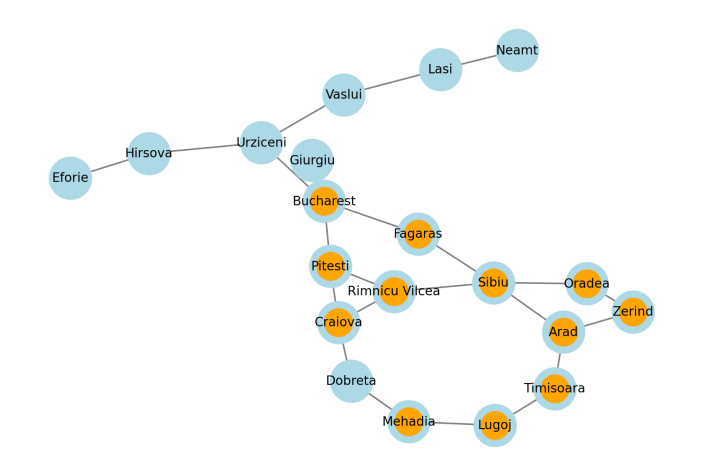
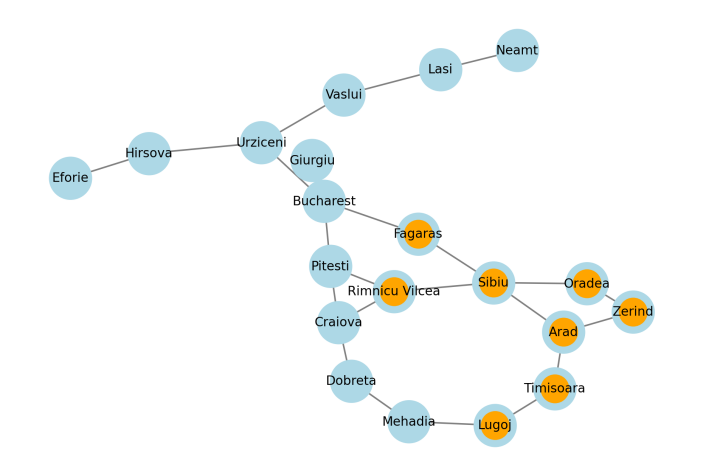
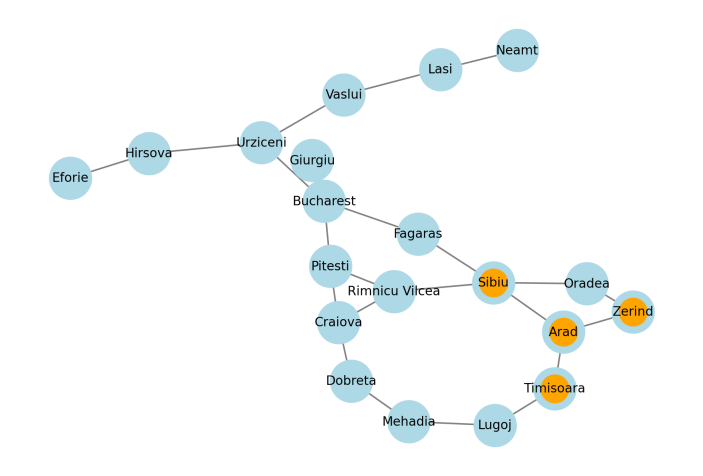
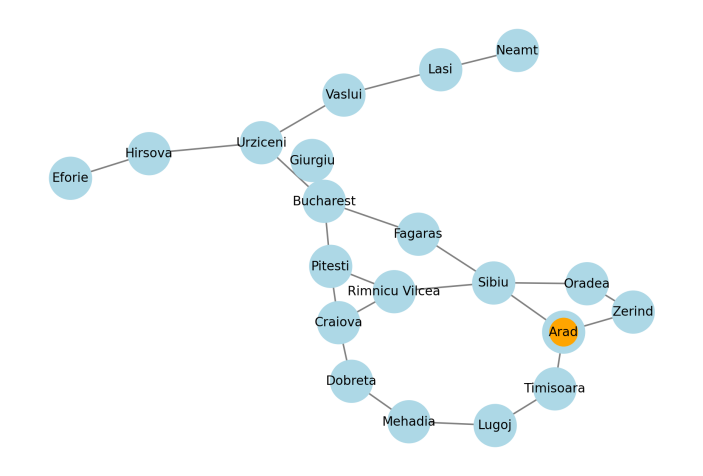
visited.ADD(neighbor)

queue.ENQUEUE(neighbor)

parent[neighbor] = current # 记录前驱节点

RETURN NULL # 未找到路径

1. **实验结果：**



1. **UCS**

**1.问题描述以及UCS理解：**

UCS：一致代价搜索，相当于BFS的加权图版本，优先扩展当前路径成本最低的节点。

用UCS实现Romania从Arad到Bucharest的路径搜索，能找到最优解。

**2.伪代码：**

FUNCTION UCS\_SHORTEST\_PATH(graph, start, end):

# 初始化优先队列（最小堆），元素格式为 (累计代价, 当前节点, 路径)

priority\_queue = MIN\_HEAP()

priority\_queue.INSERT((0, start, [start]))

visited = EMPTY\_SET() # 记录已访问节点

min\_cost = INFINITY # 初始化最小代价为无穷大

WHILE priority\_queue IS NOT EMPTY:

# 取出当前累计代价最小的节点

(current\_cost, current\_node, current\_path) = priority\_queue.POP\_MIN()

# 检查是否已访问过该节点（可能因不同路径多次入队）

IF current\_node IN visited:

CONTINUE

# 标记节点为已访问

visited.ADD(current\_node)

# 检查是否到达终点

IF current\_node == end:

RETURN current\_path, current\_cost # 返回路径和总代价

# 遍历所有邻居节点

FOR (neighbor, edge\_weight) IN graph[current\_node]:

IF neighbor NOT IN visited:

# 计算新路径的累计代价

new\_cost = current\_cost + edge\_weight

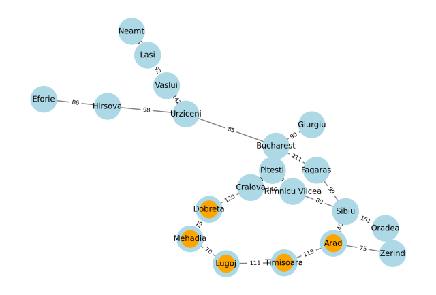
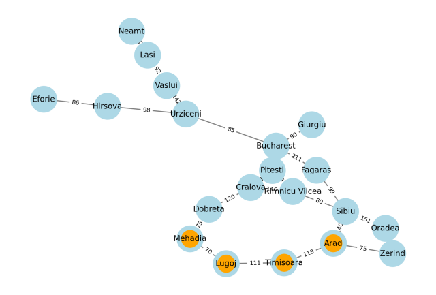
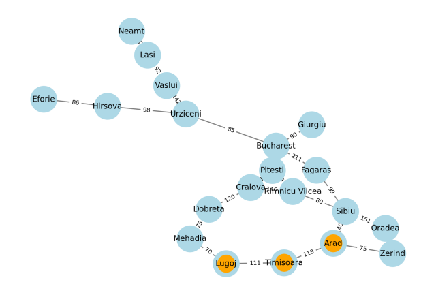
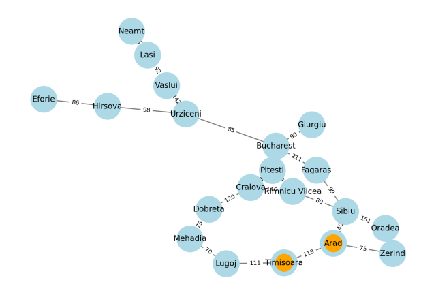
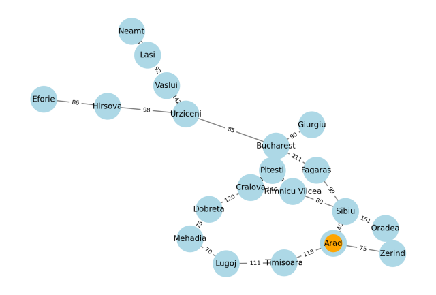
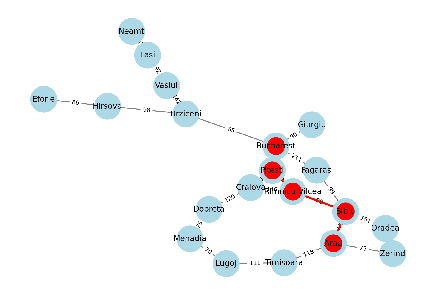
# 将新状态加入优先队列

priority\_queue.INSERT((new\_cost, neighbor, current\_path + [neighbor]))

# 如果队列为空且未到达终点，说明无法到达

RETURN NULL, INFINITY

**3.实验结果截图：**

1. **A\***

**1.问题描述以及A\*理解：**

A\*：UCS的改进版，结合路径成本（g(n)）和启发式估计（h(n)），优先扩展f(n)=g(n)+h(n)最小的节点。多用于路径规划场景。

用A\*实现Romania从Arad到Bucharest的路径搜索，能找到最优解。

**2.伪代码：**

FUNCTION A\_STAR(graph, start, end, pos):

# 输入：图 graph，起点 start，终点 end，节点坐标 pos

# 输出：从 start 到 end 的最短路径 path 及其总代价 cost

open\_set ← PRIORITY\_QUEUE() # 优先队列，按 f\_score 排序

PUSH (f\_score=HEURISTIC(start, end, pos), g\_score=0, node=start, path=[start]) INTO open\_set

visited ← DICTIONARY() # 记录已访问节点及其最小 g\_score

visited[start] ← 0

WHILE open\_set IS NOT EMPTY DO

(f, g, node, path) ← POP\_MIN(open\_set) # 取出 f 最小的节点

IF node == end THEN

RETURN path, g # 找到目标，返回路径和总代价

FOR EACH (neighbor, weight) IN graph[node] DO

new\_g ← g + weight # 计算到 neighbor 的实际代价

IF neighbor NOT IN visited OR new\_g < visited[neighbor] THEN

visited[neighbor] ← new\_g

f\_score ← new\_g + HEURISTIC(neighbor, end, pos)

PUSH (f\_score, new\_g, neighbor, path + [neighbor]) INTO open\_set

END WHILE

RETURN [], ∞ # 无可行路径，返回空路径和无限代价

END FUNCTION

# 主程序

PROCEDURE MAIN():

graph ← {

'Arad': [('Zerind', 75), ('Timisoara', 118), ('Sibiu', 140)],

'Zerind': [('Oradea', 71), ('Arad', 75)],

# ...省略其他节点定义（同原数据）

}

pos ← {

# 每个城市的二维坐标，例如：'Arad': (x, y)

}

start ← 'Arad'

end ← 'Bucharest'

path, cost ← A\_STAR(graph, start, end, pos)

IF path ≠ [] THEN

PRINT "A\*找到的最短路径：", path

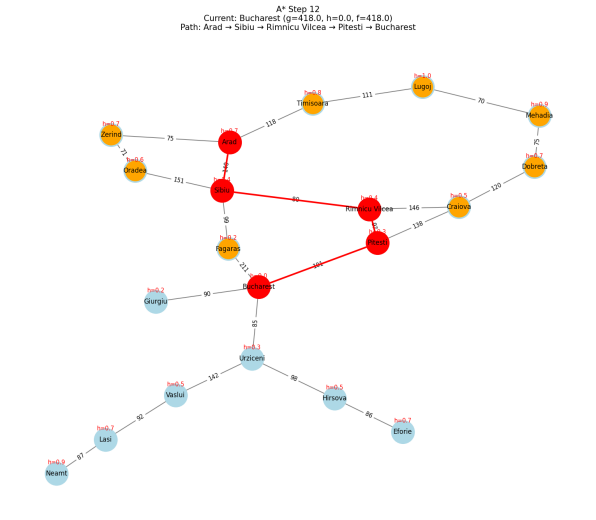
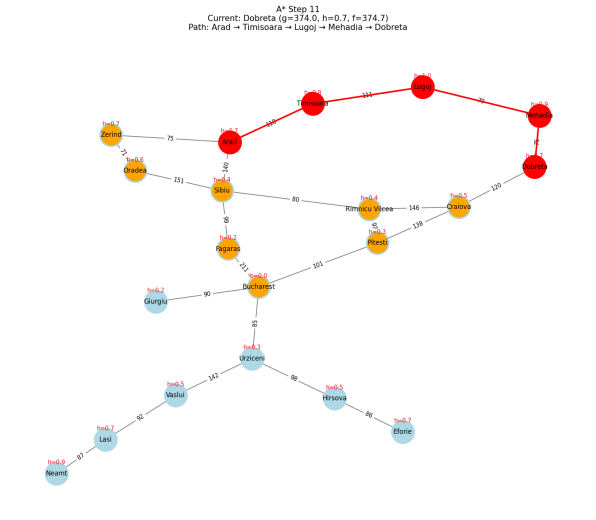
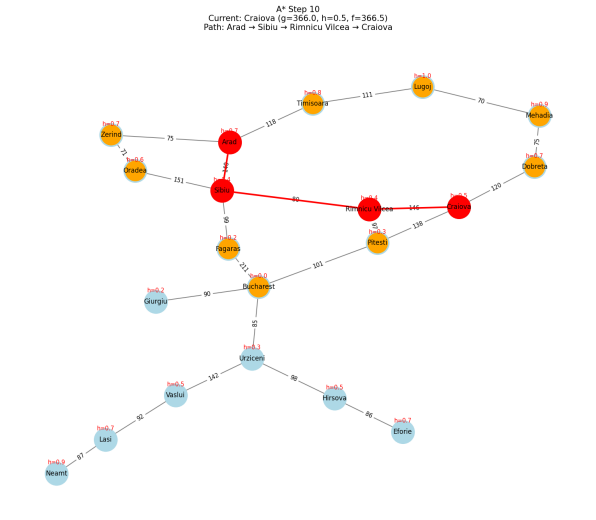
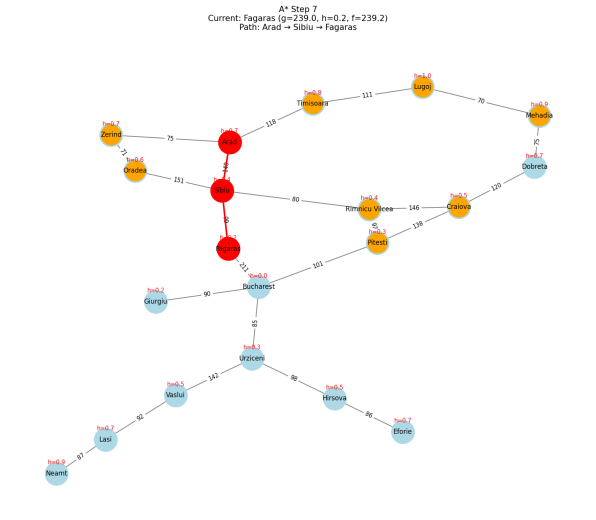
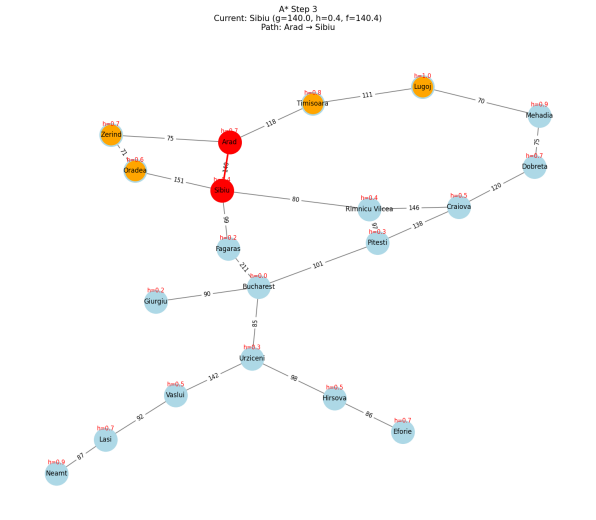
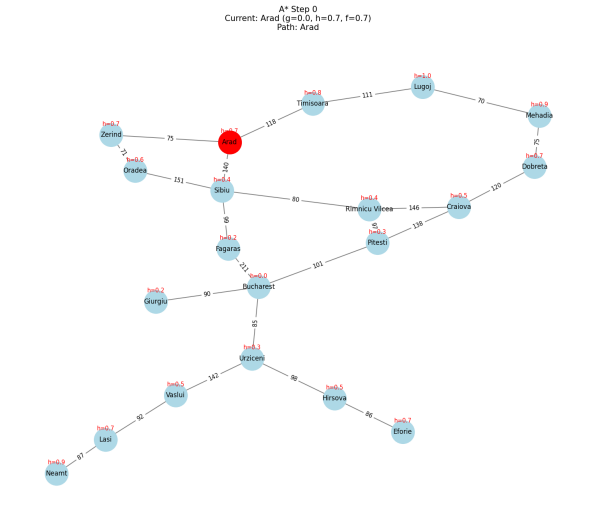
PRINT "路径总代价：", cost

ELSE

PRINT "未找到可行路径"

END PROCEDURE

**3.实验结果：**

****

1. **GS贪心**

**1.问题描述以及GS理解：**

GS：仅基于启发式函数h(n)选择节点，忽略已积累的路径成本。速度快但可能陷入局部最优解。

用GS实现Romania从Arad到Bucharest的路径搜索。

**2.伪代码：**

FUNCTION GREEDY\_SEARCH(graph, start, end, pos):

# 输入：图 graph，起点 start，终点 end，节点坐标 pos（用于计算启发值）

# 输出：从 start 到 end 的路径 path 及其总代价 cost，若无解返回空路径和 ∞

DEFINE FUNCTION heuristic(node, end, pos):

# 计算节点 node 到目标节点 end 的欧几里得距离

RETURN sqrt((pos[node][0] - pos[end][0])² + (pos[node][1] - pos[end][1])²)

END FUNCTION

open\_set ← PRIORITY\_QUEUE() # 初始化优先队列，按启发值排序

visited ← EMPTY\_SET # 初始化已访问节点集合

PUSH (heuristic(start, end, pos), start, [start]) INTO open\_set

WHILE open\_set IS NOT EMPTY DO

(h, current\_node, path) ← POP\_MIN(open\_set) # 取出启发值最小的节点

IF current\_node == end THEN

cost ← 0

FOR i FROM 0 TO LENGTH(path) - 2 DO

current ← path[i]

next ← path[i+1]

FOR EACH (neighbor, weight) IN graph[current] DO

IF neighbor == next THEN

cost ← cost + weight

BREAK

RETURN path, cost # 找到目标，返回路径及总代价

IF current\_node IN visited THEN

CONTINUE # 如果已访问则跳过

ADD current\_node TO visited

FOR EACH (neighbor, weight) IN graph[current\_node] DO

IF neighbor NOT IN visited THEN

h\_neighbor ← heuristic(neighbor, end, pos)

new\_path ← path + [neighbor]

PUSH (h\_neighbor, neighbor, new\_path) INTO open\_set

END WHILE

RETURN [], ∞ # 没有找到路径

END FUNCTION

# 主程序

PROCEDURE MAIN():

graph ← {

'Arad': [('Zerind', 75), ('Timisoara', 118), ('Sibiu', 140)],

'Zerind': [('Oradea', 71), ('Arad', 75)],

# ... 其他节点同原数据定义

}

pos ← {

# 每个城市的平面坐标，用于启发函数（此处略，可用 spring\_layout 生成）

}

start ← 'Arad'

end ← 'Bucharest'

path, cost ← GREEDY\_SEARCH(graph, start, end, pos)

IF path ≠ [] THEN

PRINT "找到路径：", path

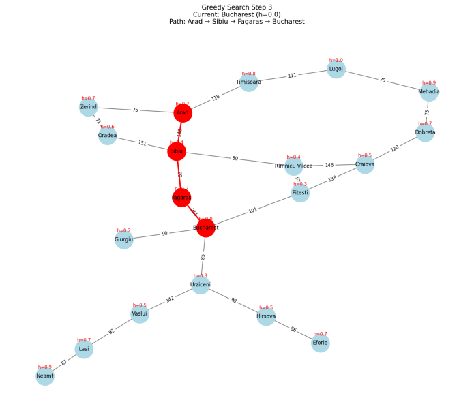
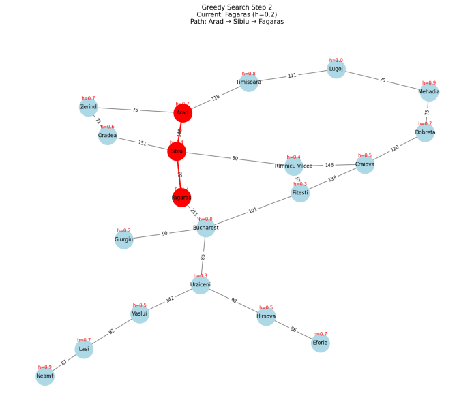
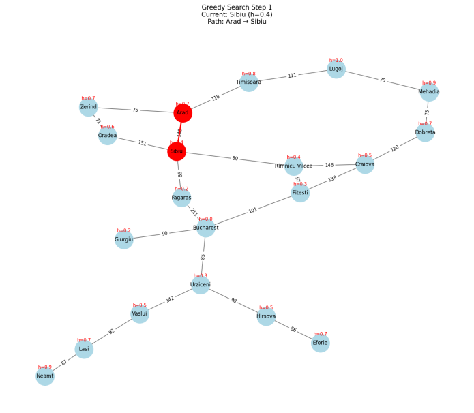
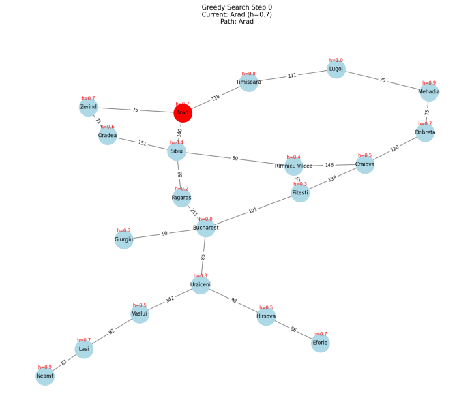
PRINT "路径总代价：", cost

ELSE

PRINT "未找到可行路径"

END PROCEDURE

**3.实验结果截图：**



对比DFS、BFS、UCS、A\*、GS：

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 最优性 | 启发式应用 | 特点 |
| DFS | 无 | 无 | 空间复杂度小（O(bm) |
| BFS | 无权图有最短路径 | 无 | 具有完备性，空间复杂度大 |
| UCS | 有 | 无 | BFS的加权图 |
| A\* | 有（h(n)可采纳） | g(n)+h(n) | UCS的改进 |
| GS | 无 | 仅h(n) | 速度快，但会陷入局部最优解 |

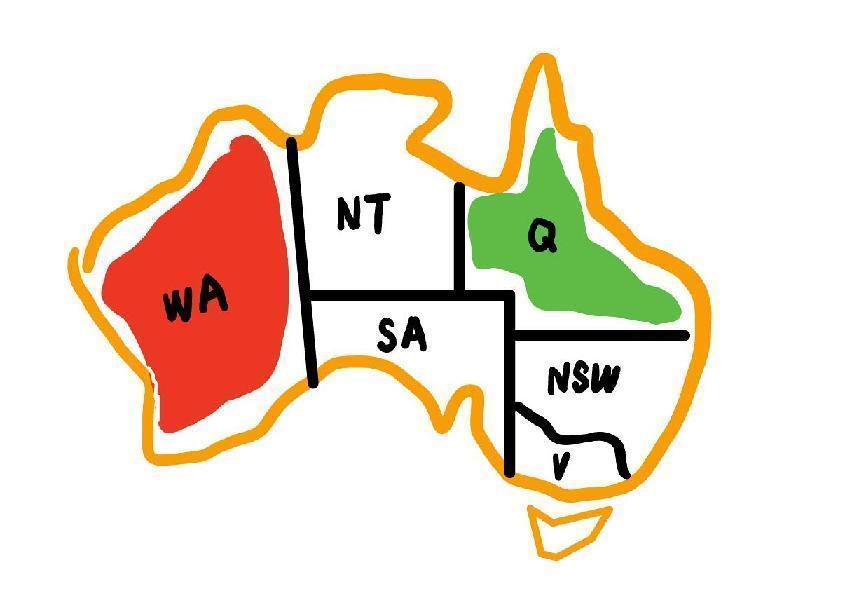
UCS、A\*都是在BFS的基础上进行改进的。

1. **AC3**

**1.问题描述以及AC3理解：**

AC3：弧一致性问题，是一种约束满足问题（CSP）的算法，用于检测和删除变量值域中不满足约束的值，使问题达到弧相容的状态。

问题描述：澳大利亚地图着色问题，设置两个初始约束:WA=red、Q=green，变量的域：NT、SA、NSW、V∈{red，green，blue}，约束：{WA≠NT;WA≠SA；NT≠SA；NT≠Q；SA≠Q;SA≠NSW;SA≠V;Q≠NSW;NSW≠V}



**2.伪代码：**

# 全局变量定义

Queue queue # 存放约束对的队列

Dictionary domains # 每个变量的取值集合（初始域）

Set variables # 所有变量名集合

Set constraints # 所有变量对之间的约束集合（邻接关系）

# 修剪函数：根据 Xi 与 Xj 的约束，修剪 Xi 的域

REVISE(Xi, Xj):

revised ← false

for each x in domains[Xi]:

# 若 Xi 中的某个值 x 与 Xj 所有可能取值 y 都冲突

if not exists y in domains[Xj] such that x ≠ y:

remove x from domains[Xi]

revised ← true

return revised

# AC-3 算法主流程

AC3():

# 初始化队列：所有约束对的两个方向都加入队列

for each (Xi, Xj) in constraints:

queue.enqueue((Xi, Xj))

queue.enqueue((Xj, Xi))

while not queue.is\_empty():

(Xi, Xj) ← queue.dequeue()

if REVISE(Xi, Xj):

# 若修剪后 Xi 的域为空，说明冲突无法满足

if domains[Xi] is empty:

return false # 无解

# 将所有与 Xi 相邻、且不等于 Xj 的变量加入队列

for each Xk in variables:

if Xk ≠ Xj and (Xk, Xi) in constraints:

queue.enqueue((Xk, Xi))

return true # 修剪完成，无冲突

# 主程序入口

Main():

# 初始化变量

variables ← {WA, NT, SA, Q, NSW, V, T}

# 初始化约束（相邻省份对）

constraints ← {

(WA, NT), (WA, SA), (NT, SA), (NT, Q),

(SA, Q), (SA, NSW), (SA, V), (Q, NSW), (NSW, V)

}

# 初始化域：WA 限定为 red，Q 限定为 green，其余为三色

domains ← {

WA: {red},

NT: {red, green, blue},

SA: {red, green, blue},

Q: {green},

NSW: {red, green, blue},

V: {red, green, blue},

T: {red, green, blue}

}

result ← AC3()

if result == true:

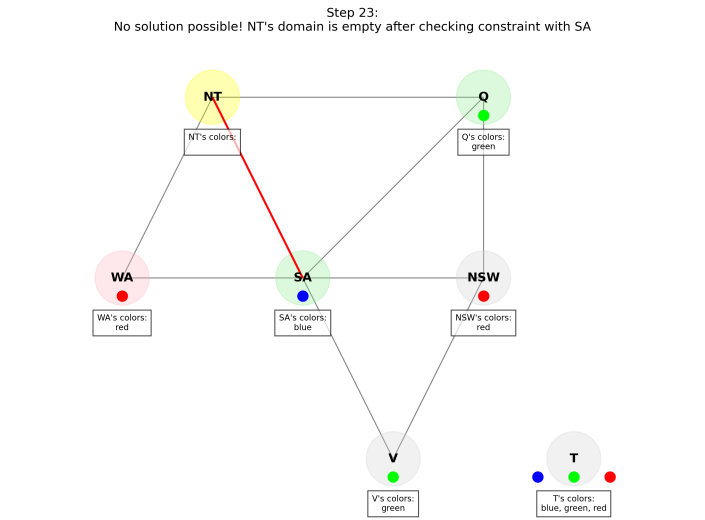
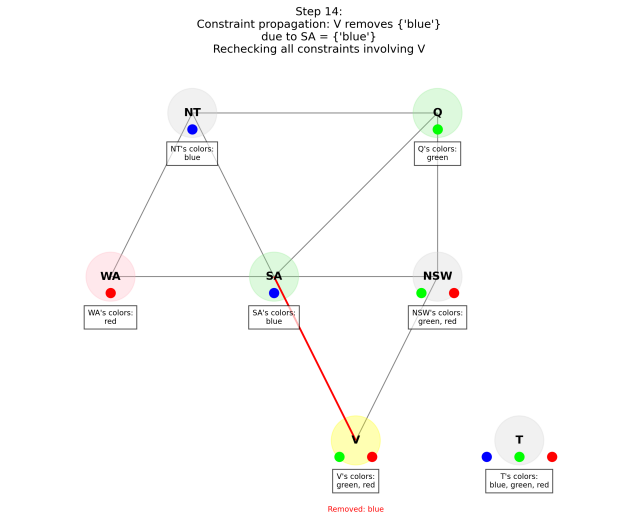
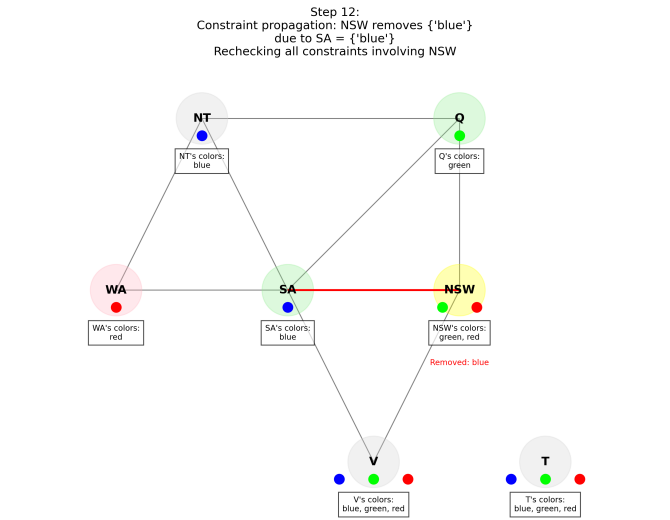
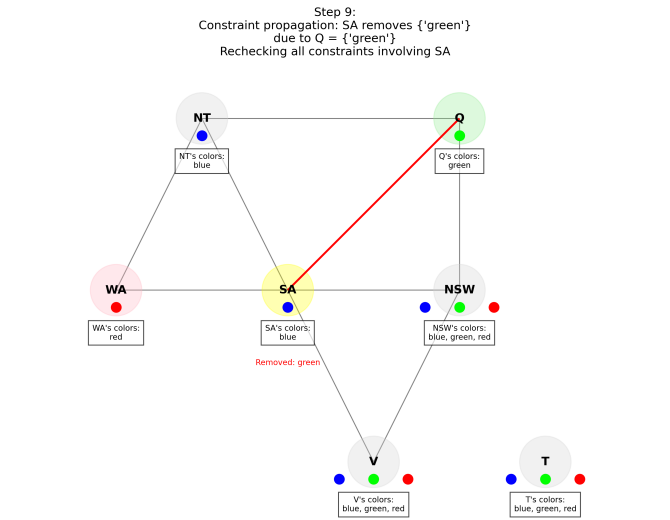
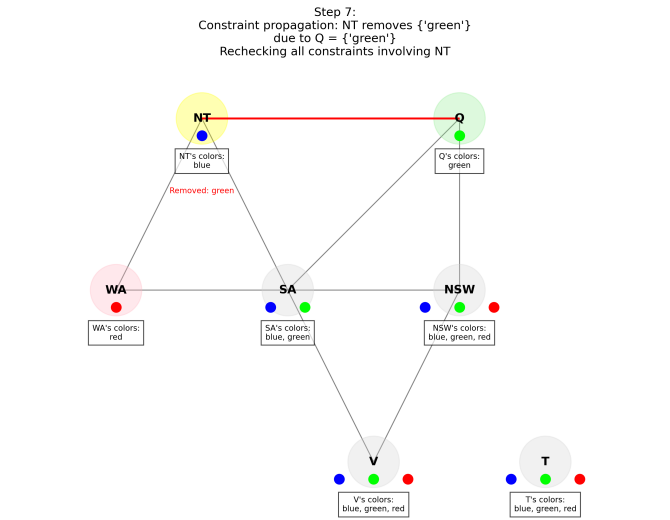
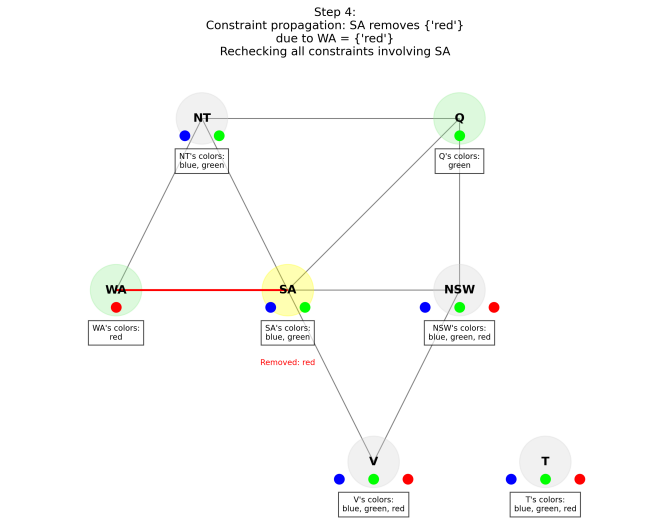
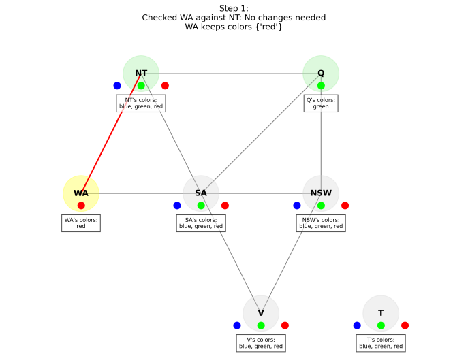
print("成功：所有变量约束一致")

print(domains)

else:

print("失败：存在变量域为空，约束冲突")

**3.实验结果：**



1. **Minimax**
2. **问题描述以及Minimax理解：**

Minimax：是一种递归的回溯算法，用于在博弈树中选择最优走法，常用于零和博弈，一方选择最大化、另一方选择最小化

问题描述：本来打算做棋类游戏（如井字游戏），但是由于算法量大，它生成的博弈树非常凌乱复杂，就放弃了这个例子），直接用这个例子来进行算法体现，也方便A-B剪枝的体现：

A

/ \

B C

/ \ / \

D E F G

/ \ / \ / \ / \

-1 4 2 6 -3 -5 0 7

1. **伪代码：**

# 全局变量定义

Dictionary tree # 树的拓扑结构，键为父节点，值为子节点列表

Dictionary leaf\_values # 叶节点的静态评估值

Dictionary node\_types # 节点类型：True 表示 Max 层，False 表示 Min 层，None 表示叶节点

Dictionary node\_values # 存储每个节点的 Minimax 计算结果

MINIMAX(node):

# 如果是叶节点，返回其静态评估值

if node\_types[node] == None:

node\_values[node] ← leaf\_values[node]

return leaf\_values[node]

# 否则，递归计算所有子节点的值

children ← tree[node]

values ← empty list

for each child in children:

value ← MINIMAX(child)

append value to values

if node\_types[node] == True: # Max 层

best ← maximum of values

else: # Min 层

best ← minimum of values

node\_values[node] ← best

return best

# 主程序入口

Main():

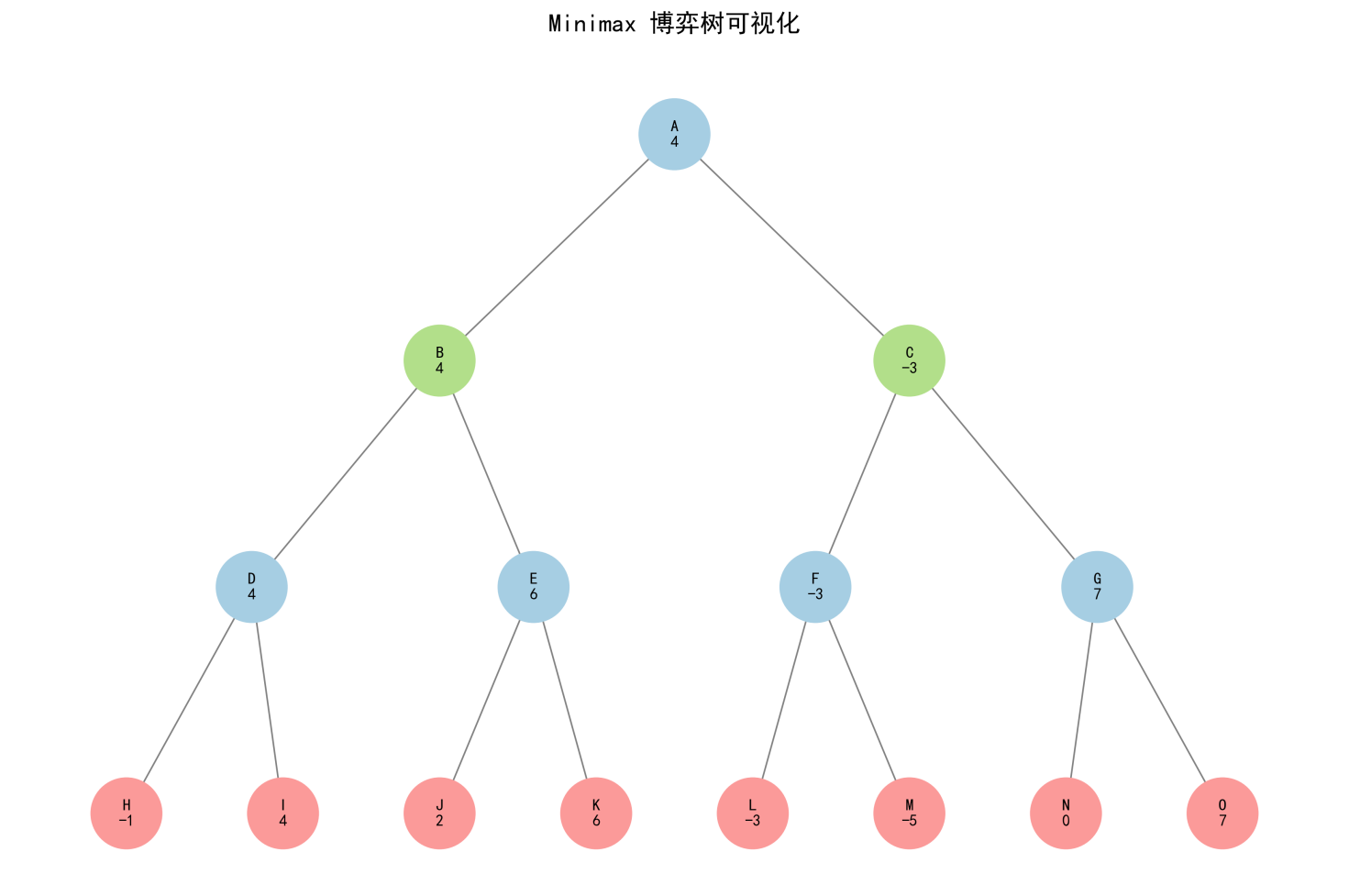
tree #初始化树的结构

leaf\_values #初始化叶结点的静态评估值

node\_types #初始化节点类型

root\_value ← MINIMAX('A') #执行minimax算法

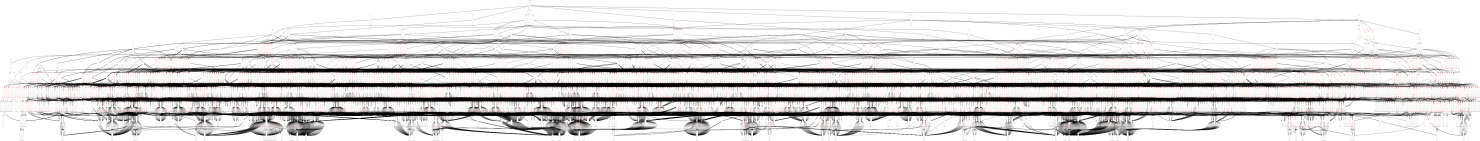
1. **实验结果截图：**



1. **α-β剪枝**
2. **问题描述以及α-β剪枝理解：**

α-β剪枝：优化minimax算法，引入两个值：α 表示当前 Max 层已知的最高下界，β 表示当前 Min 层已知的最低上界。当在某一分支上发现 α ≥ β 时，便可停止对该分支的进一步搜索，因为该分支不可能影响根节点的最终决策。最终结果会返回与原始 Minimax 相同的最优走法，但通过“剪掉”不必要的分支显著减少了节点评估量。

问题描述：为了体现于Minimax的差异，采用与上述一样的例子来体现。（同样棋类游戏最终生成的博弈树很凌乱，如下图：



）

1. **伪代码：**

# 全局变量定义

Dictionary tree # 树的拓扑结构，键为父节点，值为子节点列表

Dictionary leaf\_values # 叶节点的静态评估值

Dictionary node\_types # 节点类型：True 表示 Max 层，False 表示 Min 层，None 表示叶节点

Dictionary node\_values # 存储每个节点的 Minimax 计算结果

Dictionary alpha\_values # 存储每个节点的 alpha 值

Dictionary beta\_values # 存储每个节点的 beta 值

List pruned\_nodes # 存储被剪枝的节点

MINIMAX\_AB(node, alpha, beta):

if node\_types[node] == None:

node\_values[node] ← leaf\_values[node]

return leaf\_values[node]

if node\_types[node] == True: # Max 层

value ← -∞

for each child in tree[node]:

child\_value ← MINIMAX\_AB(child, alpha, beta)

value ← max(value, child\_value)

alpha ← max(alpha, value)

if alpha ≥ beta:

pruned\_nodes.append(remaining children in tree[node] after child)

break

node\_values[node] ← value

alpha\_values[node] ← alpha

beta\_values[node] ← beta

return value

else: # Min 层

value ← +∞

for each child in tree[node]:

child\_value ← MINIMAX\_AB(child, alpha, beta)

value ← min(value, child\_value)

beta ← min(beta, value)

if beta ≤ alpha:

pruned\_nodes.append(remaining children in tree[node] after child)

break

node\_values[node] ← value

alpha\_values[node] ← alpha

beta\_values[node] ← beta

return value

# 主程序入口

Main():

tree # 初始化树的结构

leaf\_values # 初始化叶节点的静态评估值

node\_types # 初始化节点类型

node\_values ← empty dictionary

alpha\_values ← empty dictionary

beta\_values ← empty dictionary

pruned\_nodes ← empty list

root\_value ← MINIMAX\_AB('A', -∞, +∞) # 执行 Alpha-Beta 剪枝的 Minimax 算法 **3. 实验结果截图**

